JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 8, November-December 2008

Converting Specifications in a Subset of Object-Z to Skeletal Spec# Code for both Static and Dynamic Analysis

Xiufeng Ni and Cui Zhang
Department of Computer Science, California State University, Sacramento

Abstract

Construction of correctness is an essential issue for the implementation of a reliable software system. Formal methods based verification techniques provide programmers various ways to reason their program correctness through mathematically supported static analysis and dynamic analysis. In this paper, we introduce a tool that converts formal specifications in a subset of Object-Z to skeletal Spec# code with assertions. This tool aims at facilitating the refinement from formal specifications to Spec# and the full usage of the static and dynamic analysis techniques in Spec#.

1 INTRODUCTION

It is a challenge to deliver a reliable software system while the system is large and complex. The major part of effort is spent in ensuring program correctness. In general, a program is correct and perceived reliable when the results of execution are the same as the program's intended meaning which is defined in the requirement specifications.

Formal methods refer to mathematically rigorous techniques and tools for the specification, design and verification of software systems [1]. The principles of formal methods based Correctness by Construction [2] help ensure that errors are not introduced into and can be removed from the software development process. Correctness by Construction strongly supports a formal development from requirement specification to system implementation, offering a logical solution to the problem of software reliability [3].

Object-Z is an extension of Z, one of the most widely used formal specification notation [4]. The object-oriented styled Object-Z makes it possible to structure and develop large and complex software system specifications by defining a system as collections of independent classes and objects. A seamless development by using an object-oriented approach to specification and then implementation can reduce the chance of error.

Formal methods supported verification involves static and dynamic analysis techniques. One successful dynamic technique is Design by Contract, which was introduced by Bertrand Meyer for Eiffel [5]. Using contracts, a set of assertions, Design by Contract can facilitate the automatic dynamic checking of assertions to guarantee that the system in the development can perform in accordance with its specifications. However, dynamic analysis alone can not guarantee software with reliability. Static techniques are the most rigorous methods for analyzing source code and verifying program correctness without executing the program. Static techniques and dynamic techniques should and can complement each other to ensure software reliability.

In this paper, we introduce our Object-Z-to-Spec# system, a tool that can capture formal requirement specifications in Object-Z and translate them into skeletal Spec# code to guide actual implementations. In this way, the semantic gap between Object-Z specification and C# programs with logic assertions can be bridged, the advantages of Object Z can be fully utilized, and both the dynamic and static verification tools available in Spec# can help improve code quality and the reliability of systems developed. Section 2 of this paper discusses the background and related work. Section 3 presents the Object-Z-to-Spec# system architecture and design methodology. An example of CreditCard is provided in Section 4 to illustrate how the system works. Section 5 concludes the paper and points out the potential future work to further enhance this system.

2 BACKGROUND AND RELATED WORK

Design by Contract

It has been evidenced that Design by Contract has positive effect on design, testing, and debugging activities [6]. Considering the advantages of Design by Contract, there have been efforts to incorporate this technique into various object-oriented programming languages. Contract Sharp [7] is one of the examples. Contract Sharp incorporates Design by Contract to C# programming language. The system has a syntax directed graphical user interface and generates run time traces of program executions reflecting the automated dynamic checking of assertions. Using XML, Contract Sharp allows programmers to go back and modify the source code without requiring the contracts to be recreated. Contract Sharp also automatically generates documentation.

The limitation of the Design by Contract technique available in programming languages is that it lacks a connection with formal specifications at a level much higher than programming languages. In addition, Design by Contract belongs to the family of dynamic analysis techniques thus does not support static analysis.

Object-Z to Java Skeletal Code

As the well accepted formal notation and techniques, various tools have been developed to support for Object-Z and Design by Contract. Rafsanjani and Colwill [8] describe a structural mapping from Object-Z to C++ using an object-model to reconcile the structural aspects of Object-Z and C++. Clemens Fischer from University of Oldenburg, Germany, did work on the conversion of Object-Z and CSP, another specification language, to Java in 1999 and 2000 for his PhD thesis [9]. Ramkarthik and Zhang developed a system that generates Design by Contract supported Java skeletal code from Object-Z specifications [10]. This system provides a direct one to one correlation between components at Z level and at Java level and aids in easy understanding of the individual formal specification components at both levels. Java originally had very limited built-in support for design contracts. This translation made available to Java programmers the automated dynamic verification of preconditions, post conditions. Implementing the translated Java skeletal code with dynamically checkable contracts will help improve the quality of Java classes. But there is still no direct support for static verification, which can significantly complement dynamic verification.

Static Verification, SPARK Ada, and Spec#

Static verification refers to formal method supported static analysis that does not require a program to be executed. With tool support, static verification can automatically or semi-automatically analyze the entire system. It can generate charts and reports that graphically present the analysis results, and provide recommendations of potential resolutions to identified problems [11]. The main static analysis techniques and methods include flow analysis, path function analysis, and formal program verification. Flow analysis includes control flow analysis, dataflow analysis, and information flow analysis.

There are few tools available for automated or semi-automated static verification. SPARK Ada is the most successful one. SPARK Ada approach has convinced people that rigorous static analysis should and can be an integral part of a development process [12]. Spec#, being developed by Microsoft Research Teams, also provides automated static verification technique.

SPARK Ada is a subset of Ada for high integrity programming, first formalized by Bernard Carre and Trevor Jennings of Southampton University in 1988 [13]. It has continually evolved and nowadays is being more widely used and gaining general acceptance particularly as its tools run within safety critical avionics programs. SPARK Ada now supports tagged types, tasking, and proof of exception freedom which has particular benefits in the context of RTCA DO- 178B [14]. Control flow analysis is integrated into the SPARK Ada grammar. Data and information flow analysis have been expressed as SPARK Ada program design rules. Praxis Critical Systems has used SPARK Ada in the development of several industrial programs, and their measurements indicate that the rigor provided by SPARK can be cost effective [14]. SPARK Ada offers a selection of static tools, from lightweight sanity checking to full program verification with an interactive theorem prover. SPARK Ada program can be compiled by any standard Ada compiler. However, compatibility with other existing programming languages has been an issue for SPARK Ada to be widely accepted.

Along the same line as SPARK Ada, Microsoft Research Teams have designed Spec#, a super set of the existing language C#, aiming at supporting easy and gradual adoption of dynamic and static verification features [15]. Spec# is integrated into the existing industrial-strength platform, the .NET platform. Spec# programming language consists of specification constructs like pre- and post- conditions, class invariants, and some mechanisms for higher-level data abstractions. Like Design by Contract, in Spec#, specification is part of the program, where they are checked dynamically. The Spec# programming system also consists of an automated program verifier, Boogie, which checks specifications statically. The Boogie architecture includes design-time feedback, distinct proof obligation generation and verification phases, and abstract interpretation and verification condition generation. By providing design-time feedback, Boogie moves the program verifier close to developers. By having an intermediate programming notation, BoogiePL, Spec# separates the semantic encoding of the source program from the analysis of this encoding. Comparing with SPARK Ada, Spec# is closer to major software developers.

A growing number of researchers have realized that formal specification, unambiguous languages, and both static analysis and dynamic analysis all help improve code quality and reliability. With the emerging of Spec#, in addition to the dynamic analysis techniques, the rigorous static verification techniques become available in this commonly used programming language to enhance software productivity and reliability. If there is a tool that can directly translates Object-Z specification to Spec#, C# developers can be further benefited from formal methods supported techniques at both specification language level and implementation language level. In this way, the reliability of software systems developed in .NET platform can be enhanced. This is the purpose of this project.

3. THE OBJECT-Z-TO-SPEC# SYSTEM

Project Approach

We take an approach similar to that for Object-Z Contract in Java [10]. The Object-Z-to-Spec# System is accomplished by providing a GUI to accept and facilitate formal specification in Object-Z, converting the Object-Z class schemas to XML representations, and finally generating the Spec# skeletal code through processing of the XML representation.

Object-Z-to-Spec# contains a graphical user interface for users to input specifications for classes and methods. Software engineers trained in formal specification can provide the Object-Z specification through this interface. The Object-Z specifications entered through GUI is stored as XML documents. Spec# skeletal code can be generated from XML documents according to its syntactic definition. XMLReader served as a parser is offered by the .NET platform. Also, an Object-Z style class schema can be generated from the stored XML documents. This document provides software engineers with a function of class-by-class based future review.

Spec# skeletal code generated through this process keeps most of the information from the Object-Z class schema. It includes the state variables with its types, class constants and initial schema in addition to class invariants, and all input, preconditions and postconditions for operations. This is a significant improvement over the approaches that exclude the formal specifications as part of software development process.

System Architecture

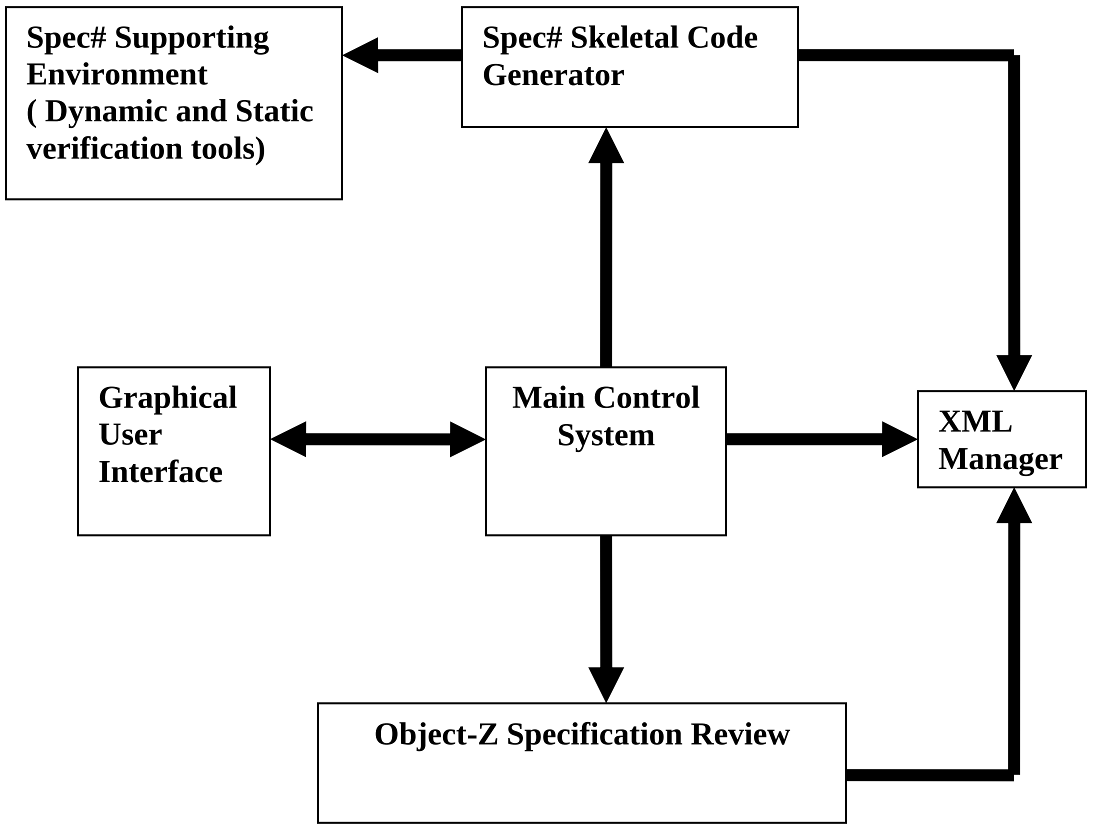

Spec# Supporting Environment (Dynamic and Static verification tools)
Spec# Skeletal Code Generator
Graphical User Interface
Main Control System
XML Manager
Object-Z Specification Review

Figure 1 Software Architecture of Object-Z-to-Spec# System

Figure 1 depicts the software architecture of the Object-Z-to-Spec# system. There are six main components in the system. The functionalities of these components are as follows:

- **Main Control System.** This is the control unit for the Object-Z-to-Spec# System. The Main Control invokes various components based on user input; captures output and passes it to appropriate components. The Main Control communicates with the Graphical User Interface (GUI). If the user action is to create the new class specification, the user will be presented with various screens to input the specification details. Once the user enters the various fields in the GUI to form a class schema specification, the Main Control passes the control to the XML Manager for further processing. If the user action is to generate Spec# skeletal code, the Main Control passes the control to the Spec# Skeletal Code Generator. If the user action is to generate a document of Object-Z specification, the Main Control passes the control to the component for Object-Z Specification Review.

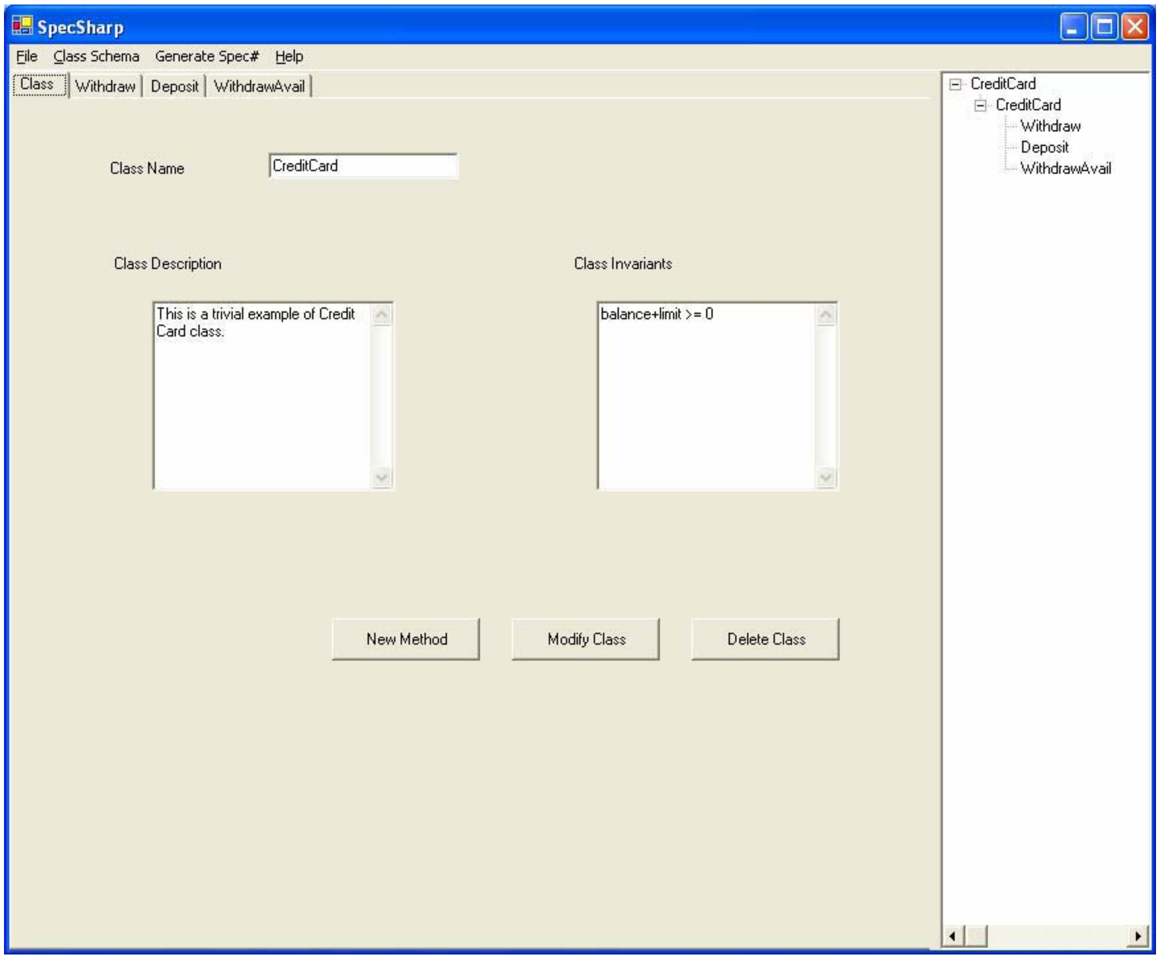
- **Graphical User Interface (GUI).** It captures the user inputs and passes them to the Main Control and presents outputs to the user. The User interface is designed in .NET platform with C#. The interface has a menu structure and a tree view. The individual menu items in the menu structure are used to input appropriate Object-Z specifications. For example choosing one menu item will help in selecting a particular class schema and its details. The user may select an Object-Z class name to be used to generate the Spec# skeletal code.
- **XML Manager.** This unit manages the process of Object-Z specification to Spec# skeletal code. It generates the XML document, maintains it, and performs XmlReader and XmlWriter on the XML document for generating the objective document. The XML Manager includes a DTD file used to define the legal building blocks of an XML document. A DTD is declared in XML document as an external reference. The XML Manager uses a pre-specified DTD file to proof check the well formalness and validity of the created XML document. When generating XML documents, the XML Manager creates a XML file in the client's machine. The XML document contains the Object-Z specification. The .NET platform provides all functions to insert, delete, read and write XML documents. XML offers important capabilities like reuse of information, information harvesting, and fine-granularity text-management applications. Thus, the Object Z specification in the form of XML document can be used for harvesting information to build Spec# skeletal codes and is reusable. The XML document is generated by XML Manager as per the document creation rules specified in DTD files.
- **Spec# Skeletal Code Generator.** The Spec# Skeletal Code Generator, with the help of the XML Manager, generates the Spec# skeletal code. The outputs from the Spec# Skeletal Code Generator are Spec# files. A Spec# file contains one or more Spec# class skeletal code that has the dynamically checkable contents extracted from the Object Z specification. It also contains the comments that a programmer can use for further implementation of the class. The generated Spec# files can be added to a Spec# project in .NET platform. So, both the dynamic and static verification tools can be available for C# programmer.
- **Object-Z Specification Review.** The outputs of Object-Z specification Review provide a class-by-class based review for the specification input by the user.

4. THE IMPLEMENTATION OF OBJECT-Z-TO-SPEC# SYSTEM

This Object-Z-to-Spec# System is developed in Microsoft's Visual Studio 2005 .NET platform. To make Spec# work in this platform, Spec# 1.0.6526 needs to be installed. Before you can use the Spec# Program Verifier, Boogie needs the file Simplify-1.5.4.exe.win in ESC/Java to be installed in Spec#. This section describes how the Object-Z-to-Spec# System is implemented, including its inputs and outputs, its functions and how they work together. The subsections below show an example of Spec# skeletal code generation from a subset of Object-Z specification for a simple CreditCard application. This simple application is refined based on [16].

Object-Z-to-Spec# GUI Functions

Figure 2 depicts the main window of the Object-Z-to-Spec# GUI with the Class Tab visible. It contains the CreditCard class in Object-Z.



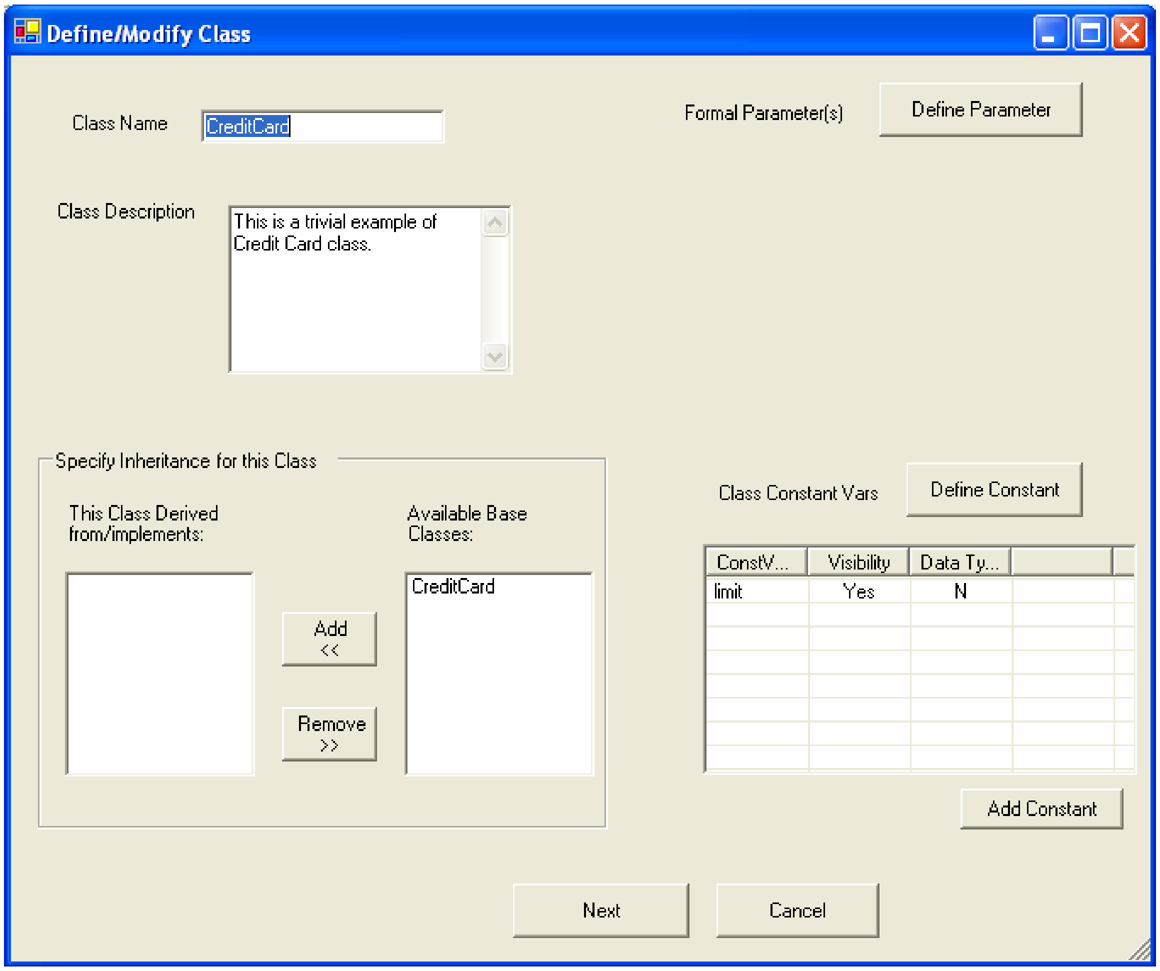
Class Name: CreditCard

Class Description: This is a trivial example of Credit Card class.

Class Invariants: balance+limit >= 0

Figure 2 Main Window of Object-Z-to-Spec# System

The Class Definition Window Form1 appears when the user clicks Modify Class or selects New→Class from the File menu. This is the first form for specifying the Object-Z class. Figure 3 shows an example for specifying the CreditCard class name, class description, inheritance, formal parameter(s), and class constants.



Class Name: CreditCard

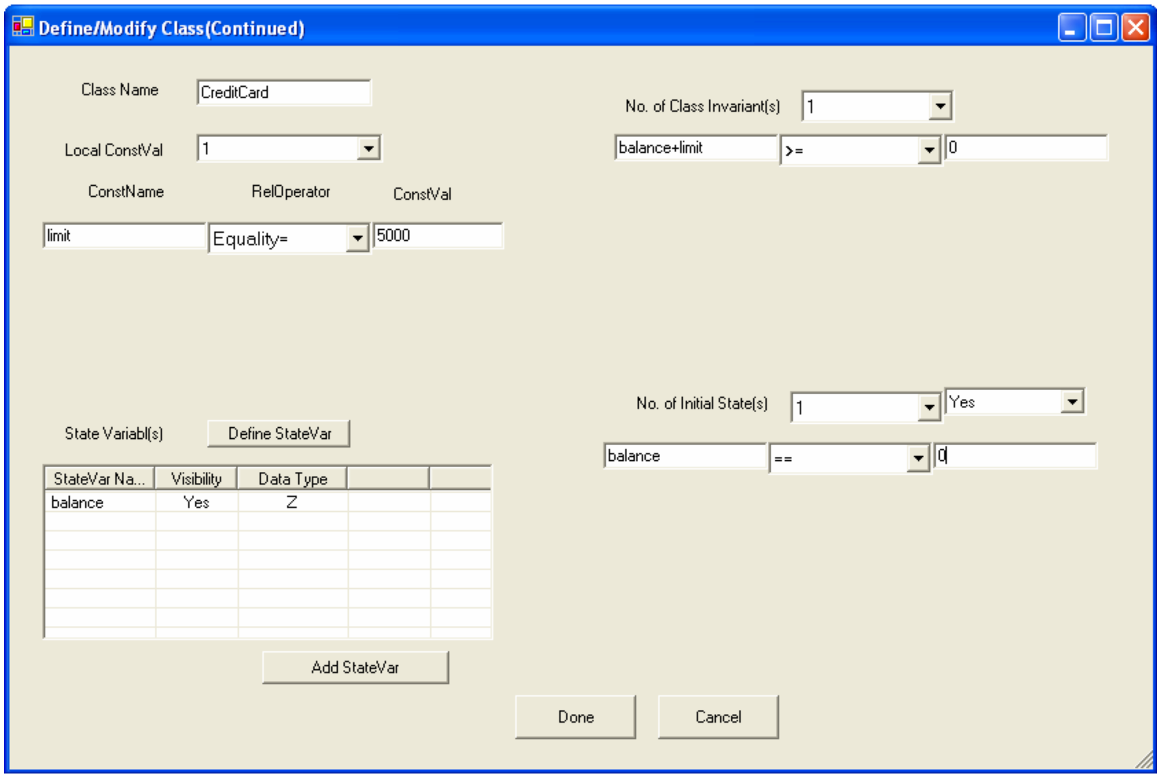
Class Description: This is a trivial example of Credit Card class.

Available Base Classes: CreditCard

ConstV...	Visibility	Data Ty...
limit	Yes	N

Figure 3 Class Definition Window Form1

After the user clicks the button Next in the Figure 3 Class Definition Window Form1, Figure 4 Class Definition Window Form2 will be shown. This is where the class constant values, class state variables, class invariants and initial schema are specified.



Class Name: CreditCard

Local ConstVal: 1

ConstName	RelOperator	ConstVal
limit	Equality=	5000

No. of Class Invariant(s): 1

balance+limit	>=	0

No. of Initial State(s): 1 Yes

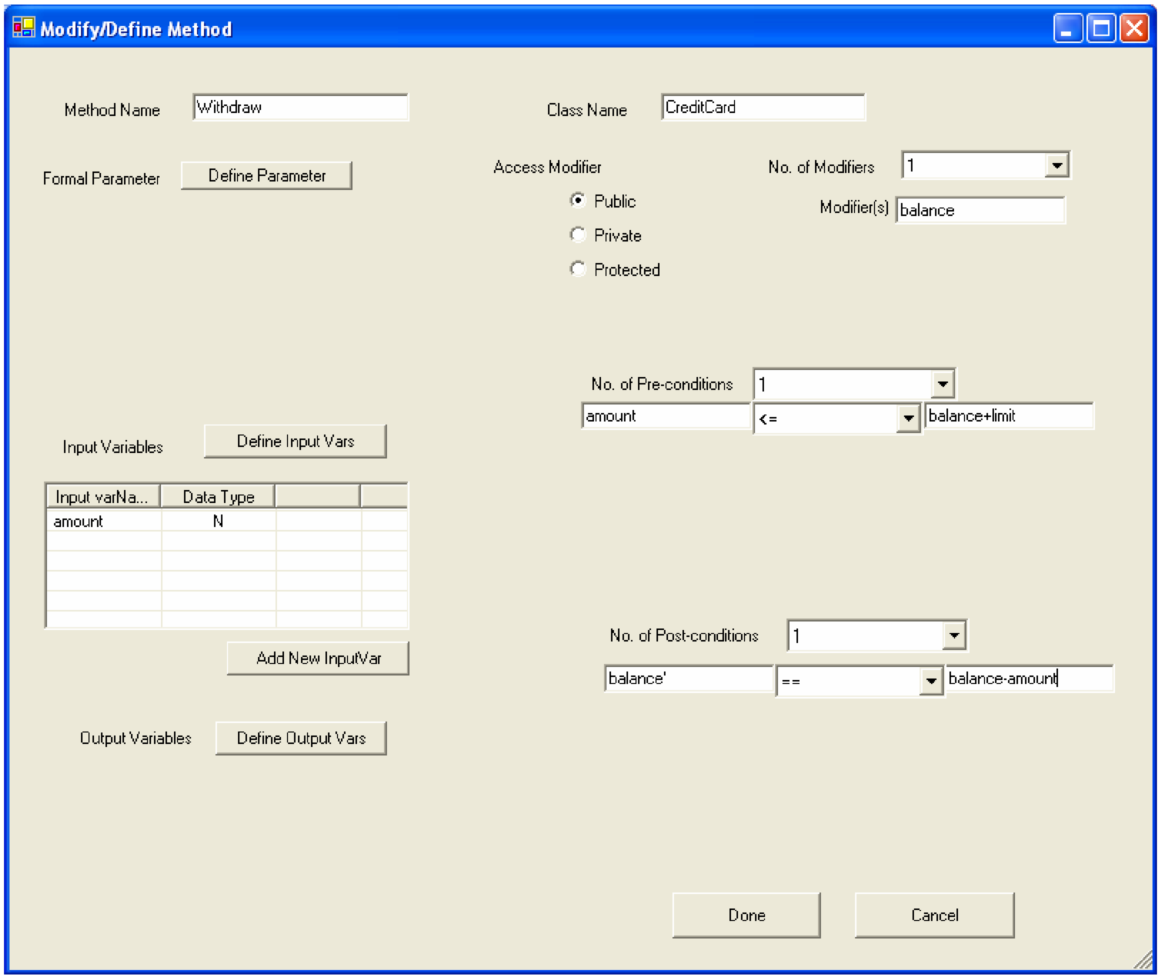
balance	==	0

State Variabl(s)

StateVar Na...	Visibility	Data Type		
balance	Yes	Z		

Figure 4 Class Definition Window Form2

The Method Definition Window Form is where the user specifies the method signature, contracts, and input, output variables. Figure 5 depicts the Method Definition Window, containing the CreditCard Withdraw method.


Modify/Define Method

Method Name: Withdraw
Class Name: CreditCard
Formal Parameter
Access Modifier: [✓ Public] Private Protected
No. of Modifiers: 1
Modifier(s): balance
No. of Pre-conditions: 1
amount <= balance+limit
Input Variables

Input varNa...	Data Type
amount	N

No. of Post-conditions: 1
balance' == balance-amount
Output Variables

Figure 5 Method Definition Window Form

The Outputs from the Illustrated Example

This section contains the running output of the CreditCard example illustrated. The outputs include one XML document for the whole project, an Object-Z specification document and the system generated Spec# skeletal code. Figure 6 contains the XML document for the CreditCard project. Figure 7 contains the Object-Z specification for the example of the CreditCard Object-Z class. Figure 8 contains the system generated Spec# skeletal code CreditCard Spec# class for the CreditCard Object-Z class.

```
<?xml version="1.0"?>
<project projname="C:\Documents and Settings\zhiyong he\My
Documents\Visual Studio Projects\CreditCard.xml">
  <date>1/4/2007</date>
  <time>10:54 PM</time>
  <class classname="CreditCard">
    <classdescription>This is a trivial example of Credit Card
class.</classdescription>
    <inheritsfrom />
    <parameters />
    <constantvars>
      <constvar>
        <name>limit</name>
        <visibility>Yes</visibility>
        <type>N</type>
        <sub>
        </sub>
      </constvar>
    </constantvars>
    <constantvals>
      <constval>limit Equality= 5000</constval>
    </constantvals>
    <statevars>
      <statevar>
        <name>balance</name>
        <visibility>Yes</visibility>
        <type>Z</type>
        <sub>
        </sub>
      </statevar>
    </statevars>
    <classinvariant>balance+limit &gt;= 0</classinvariant>
    <classinitial>
      <visibility>Yes</visibility>
      <statement>balance == 0</statement>
    </classinitial>
    <method methodname="Withdraw">
      <accessmodifier>Public</accessmodifier>
      <parameters />
      <deltalist>
        <modifier>balance</modifier>
      </deltalist>
      <inputvars>
        <input>
          <name>amount</name>
          <type>N</type>
          <sub>
          </sub>
        </input>
      </inputvars>
```

Figure 6 XML Document for CreditCard Project

```
       <outputvars />
       <precondition>amount &lt;= balance+limit</precondition>
       <postcondition>balance' == balance-amount</postcondition>
     </method>
     <method methodname="Deposit">
       <accessmodifier>Public</accessmodifier>
       <parameters />
       <deltalist>
         <modifier>balance</modifier>
       </deltalist>
       <inputvars>
         <input>
           <name>amount</name>
           <type>N</type>
           <sub>
           </sub>
         </input>
       </inputvars>
       <outputvars />
       <precondition>true</precondition>
       <postcondition>balance' == balance+amount</postcondition>
     </method>
     <method methodname="WithdrawAvail">
       <accessmodifier>Public</accessmodifier>
       <parameters />
       <deltalist>
         <modifier>balance</modifier>
       </deltalist>
       <inputvars />
       <outputvars>
         <output>
           <name>amount</name>
           <type>N</type>
           <sub>
           </sub>
         </output>
       </outputvars>
       <precondition>true</precondition>
       <postcondition>amount == balance+limit&amp;balance' == -
limit</postcondition>
     </method>
   </class>
</project>
```

Figure 6 XML Document for CreditCard Project (Continued)

```
|----------- CreditCard -------------------------------------------
|     Visibility List
|     ( limit, balance, INIT, Withdraw, Deposit, WithdrawAvail )
|
|
|     | limit : N
|     |--------------------------------------------------
|     | limit = 5000
|
|
|     |--------------------------------------------------
|     | balance : Z
|     |--------------------------------------------------
|     | balance+limit >= 0
|     |---------------------------------------------------
|
|     |--------- INIT -----------------------------------
|     | balance = 0
|     |---------------------------------------------------
|
|     |-------- Withdraw ---------------------------------
|     | delta(balance)
|     | amount? : N
|     |-------------------------------------------------
|     | amount <= balance+limit
|     | balance' = balance-amount
|     |------------------------------------------------------
|
|     |-------- Deposit ---------------------------------
|     | delta(balance)
|     | amount? : N
|     |-------------------------------------------------
|     | balance' = balance+amount
|     |------------------------------------------------------
|
|     |-------- WithdrawAvail ---------------------------------
|     | delta(balance)
|     | amount! : N
|     |-------------------------------------------------
|     | amount = balance+limit
|     | balance' = -limit
|     |------------------------------------------------------
|
|---------------------------------------------------------------
```

Figure 7 Object-Z-to-Spec# System Generated Object-Z Specification

```
using System;
using Microsoft.Contracts;
using System.IO;
using System.Text;

namespace testCreditCard
{
         public class CreditCard
         {
               [SpecPublic] int balance;
               public short limit = 5000;
               invariant limit+balance >= 0;

               public CreditCard( )
                       ensures balance == 0;
               {
               }

               public void Withdraw(short amount)
                       requires amount <= balance+limit
                              otherwise ArgumentException;
                       ensures balance == old(balance)-amount;
                       modifies balance;
               {
               }

               public void Deposit(short amount)
                       ensures balance == old(balance)+amount;
                       modifies balance;
               {
               }

               public void WithdrawAvail( short amount)
                       ensures amount == balance+limit;
                       ensures balance == amount-limit;
                       modifies balance;
               {
               }

       }//end of class
}//end namespace
```

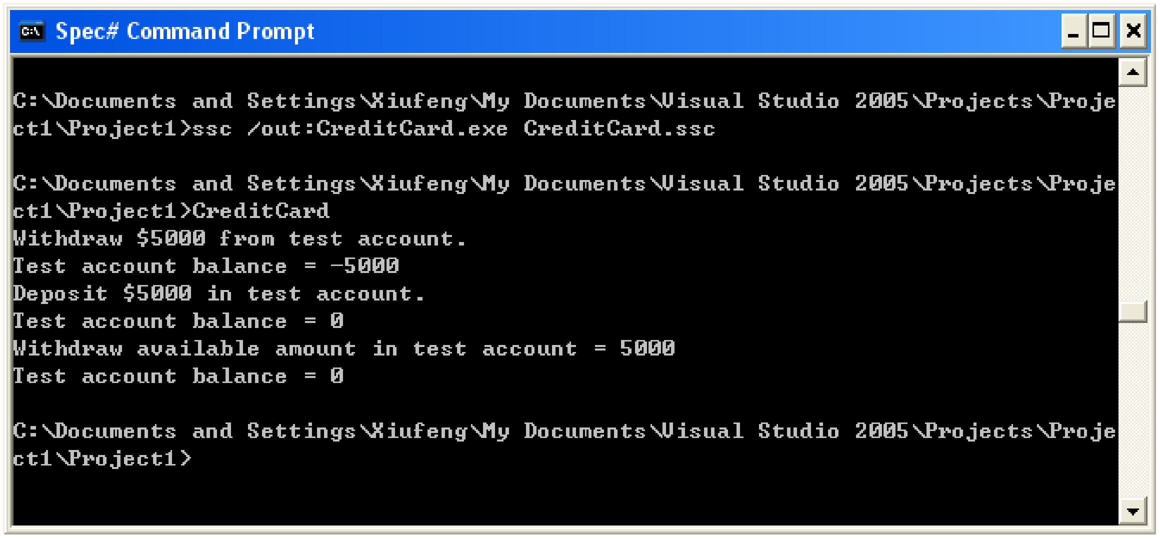
Figure 8 Object-Z-to-Spec# System Generated Spec# Skeletal Code

5. EXAMPLES OF DYNAMIC AND STATIC VERIFICATION

The previous section illustrates functionalities of the Object-Z-to-Spec# system through a small CreditCard example. Through the same example, this section shows the availability of support in Spec# for both dynamic verification and static verification. Due to the reality that the Spec# programming system is still under developing, some steps described below possibly are not stable.

Dynamic Verification Example

To complete the implementation with the Object-Z-to-Spec# system generated Spec# skeletal code, the programmer needs to create a new Spec# project in Visual Studio 2005 for adding implementation details to the skeletal Spec# code. For dynamic verification in Spec#, the programmer only needs to compile and run the implemented Spec# class. Figure 9 contains the Spec# Command Window for dynamic verification. The verified CreditCard Spec# code is contained in Figure 10.



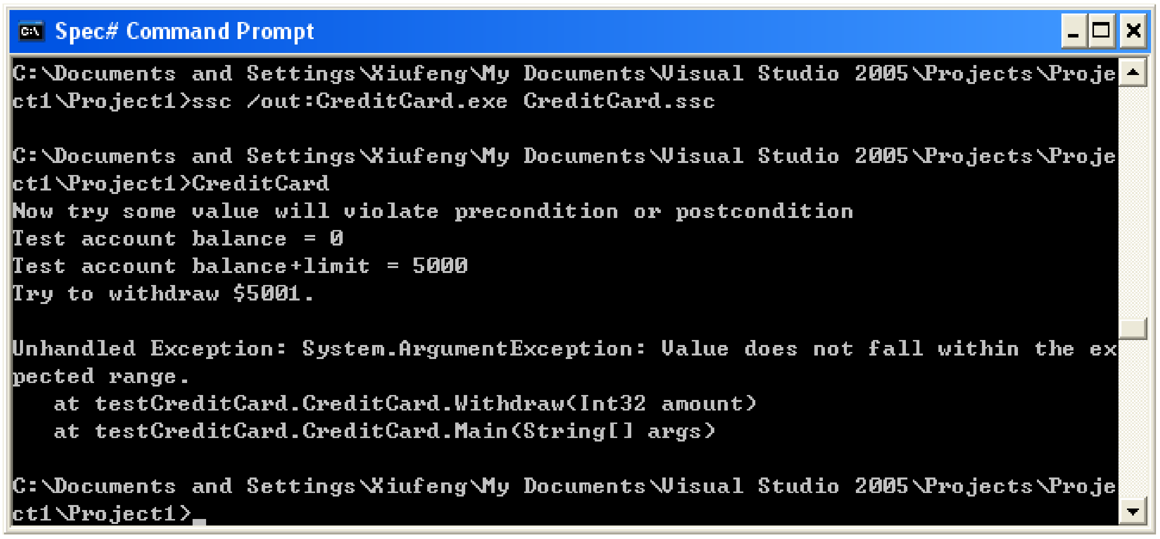
```
C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>ssc /out:CreditCard.exe CreditCard.ssc

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>CreditCard
Withdraw $5000 from test account.
Test account balance = -5000
Deposit $5000 in test account.
Test account balance = 0
Withdraw available amount in test account = 5000
Test account balance = 0

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>
```

Figure 9 Spec# Command Window for Dynamic Verification

```
using System;
using Microsoft.Contracts;
using System.IO;
using System.Text;
namespace testCreditCard
{
       public class CreditCard
       {
           [SpecPublic] int balance;
           private int avail;
           public short limit = 5000;
           invariant balance+limit >= 0;

           public CreditCard( )
                   ensures balance == 0;
           { balance = 0;}

           public void Withdraw(int amount )
                   requires amount <= balance+limit otherwise ArgumentException;
                   ensures balance+limit >= 0;
                   ensures balance == old(balance)-amount;
                   modifies balance;
          { balance = balance - amount;}

           public void Deposit(int amount )
                   ensures balance == old(balance)+amount;
                   modifies balance;
           { balance += amount; }

          public void WithdrawAvail( int amount)
                   ensures amount == balance+limit;
                   ensures balance == amount-limit;
                   modifies balance;
         {
              amount = balance + limit;
              balance = amount-limit;
              avail = amount;
         }

         public static void Main (string[] args)
         {
              CreditCard test = new CreditCard();
              test.Withdraw(5000);
              Console.WriteLine("Withdraw $5000 from test account.");
              Console.WriteLine("Test account balance = " +
                                           test.balance.ToString());
              test.Deposit(5000);
              Console.WriteLine("Deposit $5000 in test account.");
              Console.WriteLine("Test account balance = " +
                                          test.balance.ToString());
              test.WithdrawAvail(test.avail);
              Console.WriteLine("Withdraw available amount in test
                                    account = " + test.avail.ToString());
              Console.WriteLine("Test account balance = " +
              test.balance.ToString());
        }
    }//end of class
}//end namespace
```

Figure 10 The DynamicallyVerified Spec# Code in Figure 9

If the programmer adds a line of code "test.Withdraw(5001)" to the Main method in Figure 10, dynamic verifier in Spec# programming system will throw an exception, and emit error messages. Figure 11 contains such dynamic verification commands and the error messages.

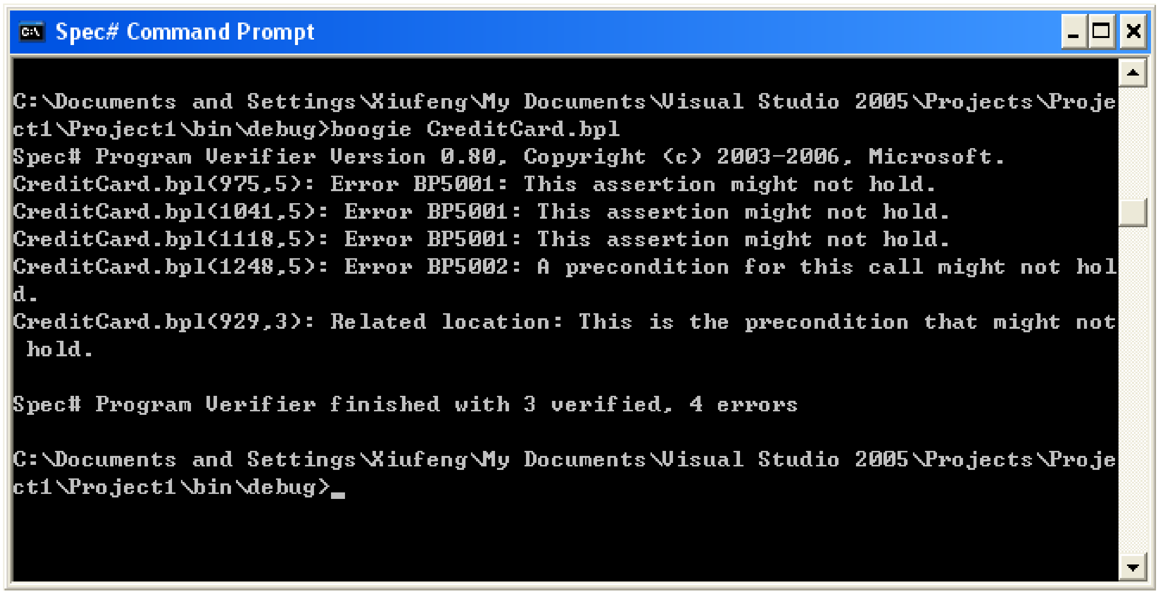

Spec# Command Prompt

```
C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>ssc /out:CreditCard.exe CreditCard.ssc

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>CreditCard
Now try some value will violate precondition or postcondition
Test account balance = 0
Test account balance+limit = 5000
Try to withdraw $5001.

Unhandled Exception: System.ArgumentException: Value does not fall within the ex
pected range.
   at testCreditCard.CreditCard.Withdraw(Int32 amount)
   at testCreditCard.CreditCard.Main(String[] args)

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1>_
```

Figure 11 Dynamic Verification with Error Messages

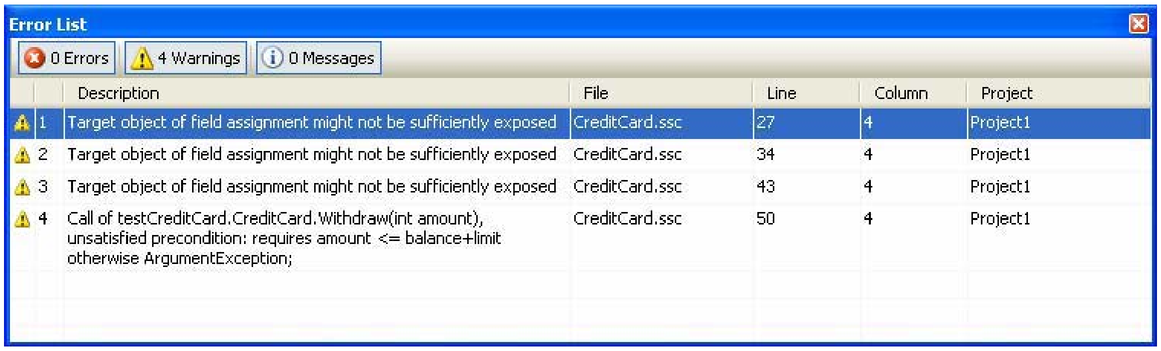
Static Verification Example

This example applies the static verifier to the previous dynamically verified Spec# CreditCard code. When compiling Spec# code, the compiler emits a BPL file for static verification. In the Spec# Command Window, the programmer can use command “boogie *.bpl”. Also, the programmer can turn on the RunProgramVerifier under the Configuration Properties in the Spec# project properties window for static verification. Or, the programmer can add a command option “\verify” to the compiling command. Figure 12 contains the Spec# static verification window with the command “boogie CreditCard.bpl”. Figure 13 contains the error information window in the Visual Studio 2005 platform with RunProgramVerifier property on. These two figures show the same errors for the Spec# code.



```
Spec# Command Prompt

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1\bin\debug>boogie CreditCard.bpl
Spec# Program Verifier Version 0.80, Copyright (c) 2003-2006, Microsoft.
CreditCard.bpl(975,5): Error BP5001: This assertion might not hold.
CreditCard.bpl(1041,5): Error BP5001: This assertion might not hold.
CreditCard.bpl(1118,5): Error BP5001: This assertion might not hold.
CreditCard.bpl(1248,5): Error BP5002: A precondition for this call might not hol
d.
CreditCard.bpl(929,3): Related location: This is the precondition that might not
 hold.

Spec# Program Verifier finished with 3 verified, 4 errors

C:\Documents and Settings\Xiufeng\My Documents\Visual Studio 2005\Projects\Proje
ct1\Project1\bin\debug>_
```

Figure 12 Static Verification of CreditCard BPL File



Error List

0 Errors | 4 Warnings | 0 Messages

		Description	File	Line	Column	Project
⚠	1	Target object of field assignment might not be sufficiently exposed	CreditCard.ssc	27	4	Project1
⚠	2	Target object of field assignment might not be sufficiently exposed	CreditCard.ssc	34	4	Project1
⚠	3	Target object of field assignment might not be sufficiently exposed	CreditCard.ssc	43	4	Project1
⚠	4	Call of testCreditCard.CreditCard.Withdraw(int amount), unsatisfied precondition: requires amount <= balance+limit otherwise ArgumentException;	CreditCard.ssc	50	4	Project1

Figure 13 Error Information Window for Static Verification

The first three warnings can be gotten rid of by placing the assignment to balance within an expose block. In the Spec# programming system, an assignment to a field of an object o is allowed only when o is exposed, i.e. in o's constructor or within an expose(o) block. Spec# separately verifies each method and only takes the specification of called methods into account, not their implementation. So, one way to get rid of the fourth warning is to add an additional postcondition to the constructor. Figure 14 contains the corrected Spec# code.

```
using System;
using Microsoft.Contracts;
using System.IO;
using System.Text;
namespace testCreditCard
{
        public class CreditCard
        {
              [SpecPublic] int balance;
               private int avail;
               public short limit = 5000;
               invariant balance+limit >= 0;
               public CreditCard( )
                    ensures balance == 0;
                    ensures limit == 0;
               { balance = 0;}
               public void Withdraw(int amount )
                    requires amount <= balance+limit otherwise ArgumentException;
                    ensures balance+limit >= 0;
                    ensures balance == old(balance)-amount;
                    modifies balance;
              { expose(this){balance = balance - amount;}}
               public void Deposit(int amount )
                    requires amount >= 0;
                    ensures balance == old(balance)+amount;
                    modifies balance;
              { expose(this){balance += amount; }}
              public void WithdrawAvail( int amount)
                    ensures amount == balance+limit;
                    ensures balance == amount-limit;
                    modifies balance;
              {
                    amount = balance + limit;
                    expose(this){balance = amount-limit;}
                    avail = amount;
              }
              public static void Main (string[] args)
              {
                    CreditCard test = new CreditCard();
                    test.Withdraw(5000);
                    test.Deposit(5000);
              }
       }//end of class
}//end namespace
```

Figure 14 the Corrected CreditCard Spec# Code

6. CONCLUSIONS AND FUTURE WORK

As Jim Woodcock advocated in [17], "Given the right computer-based tools, the use of formal methods will become widespread, transforming the practice of software engineering." To make this a reality, the computer scientists have been collaborating to develop verification technology enhancing software productivity and reliability. Spec# programming language seems a right tool to facilitate the application of formal methods as it is built upon C#, a widely used object oriented programming language integrated into the .NET platform.

Our work on the Object-Z-to-Spec# system is an attempt in the direction of making formal specification and verification available for software practitioners. By automating the translation from Object-Z specifications to Skeletal Spec# code capturing the information form Object-Z, our system bridges the semantic gap between Object-Z and Spec#, facilitates the refinement from Object-Z to Spec#, and encourages practitioners to fully take the advantages of formal specification in Object-Z and of the static and dynamic analysis techniques available in Spec#.

The Object-Z-to-Spec# system presented can be further improved. For instance, the system currently does not provide a way to handle the method return type and return value verification. This limitation should be addressed in the future. Besides, the system currently only supports formal specifications in a subset of Object-Z with primitive data types only, which limits the expressiveness of specifications in Object-Z. The system can be extended and enhanced by providing all data types for the constants and variables according to the Object-Z data type definitions. Formal specification in the full Object-Z can then be converted to skeletal Spec# code for the implementation strengthened by the formal methods supported static and dynamic analysis techniques.

REFERENCES

[1] David Crocker, Developing Reliable Software using Object-Oriented Formal Specification and Refinement, Escher Technologies Ltd., Mallard House, Hillside Road, Ash Vale, Aldershot GU12 5BJ, United Kingdom.

[2]Martin Croxford, Roderick Chapman, http://www.stsc.hill.af.mil/crosstalk/2005/12/0512CroxfordChapman.html, October 2006.

[3] Ricky W. Butler, What is formal Methods?, http://shemesh.larc.nasa.gov/fm/fm-what.html, August 2006.

[4] Graeme Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 1999.

[5] Bertrand Meyer, Eiffel: The Language, Prentice Hall, 1992.

[6] Robert Binder, “Testing Object-Oriented Systems,” Addison Wesley Longman, 2000, pp 807-916.

[7] Rachel Henne-Wu, William Mitchell, and Cui Zhang, Support for Design by Contract in the C# Programming Language, Journal of Object Technology, September-October 2004, pp65-82.

[8] GHB Rafsanjani and SJ Colwill, From Object-Z to C++: A Structural Mapping. In Z User Meeting (ZUM'92). Springer-Verlag, 1992.

[9] Clemens Fischer, Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis. University of Oldenburg, 2000.

[10] Sowmiya Ramkarthik and Cui Zhang, Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z, Proceedings of the IEEE/ACIS International Conference on Computer and Information Science, Honolulu, Hawaii, July 2006, pp. 405-411

[11] Andy German, Air Vehicle Software Static Code Analysis Lessons Learned – Ninth Safety Critical Club Symposium, Bristol, United Kingdom: Springer, February 2001.

[12] John Barnes, High Integrity Software The SPARK Approach to Safety and Security, Addison-Wesley, 2002.

[13] SPARK(programming language), http://en.wikipedia.org/wiki/SPARK_programming_language, August 2006.

[14] SPARKAda Quotes, http://www.praxis-his.com/sparkada/quotes.asp, August 2006.

[15] Microsoft Research Spec#: http://research.microsoft.com/specsharp.

[16] Brian Stevens, Implementing Object-Z with Perfect Developer, Journal of Object Technology, March-April 2006, pp189-202.

[17] Jim Woodcock, First Steps in the Verified Software Grand Challenge, IEEE Computer Society, October 2006, pp57-64.

About the authors

Xiufeng Ni earned her Master degree in Computer Science from California State University, Sacramento, in May, 2007. She is currently employed by AgileIT Inc. in White Plains, New York as a software developer. She can be reached at xiufeng@agileitinc.com.

Dr. Cui Zhang is a full professor in the Department of Computer Science, California State University Sacramento (CSUS). Her research and teaching interests include software engineering, object-oriented techniques and methodologies, programming languages, and formal methods for software engineering and for information assurance and security.She received her Ph.D. in Computer Science from Nanjing University, China, in 1986. She has published research papers in professional conferences and journals, and served on program committees for several international conferences. She can be reached at zhangc@ecs.csus.edu.